Docker监控

原创 2018-03-12 张健 [Docker](https://mp.weixin.qq.com/s?__biz=MzA5OTAyNzQ2OA==&mid=2649697153&idx=1&sn=546e58faeb8380726ae2e675abf4e754&chksm=889314e2bfe49df42246d351726e54d3d6028ecc5b0570774a97b258cd0cd541e1b4cc1d0b95&mpshare=1&scene=24&srcid=0312cLQ2vMX1wLm61y95tvuX&key=03061ca3b0ce72934191bb9fe84843cf9fed86a165d6f446c5e2e9175821bf89dd97c8380fa1aab7dd4b125336e02af5e01042f3b3ecda6feadb710c0738f983fe5a0f56443ea172a02b98d176de446a&ascene=14&uin=Nzk3NjIyNQ%3D%3D&devicetype=Windows+8&version=6206014b&lang=zh_CN&pass_ticket=lz%2FWjVtDLPWgjZvsnD4SLpRaZaVMVoSg3%2Bv7cF%2Bp3So%3D&winzoom=1" \l "#)

现在有很多的开源的Docker监控方案的实现，我们可以很容易的搭建一套监控系统出来；但是如果你有定制化的需求，则需要自己去实现；那么我们该怎么实现呢？需要监控哪些指标呢？这些指标又是什么含义呢？应该怎样去收集呢？本文我们来一起探讨。  
这里我不会介绍整个监控系统的架构，也不会去分别介绍存储、告警、展示、通知等等这些模块的实现，因为现在的开源的监控系统基本都包括这些，只会把重点放在Docker的指标上，所以内容会有些干。接触时间也不算太长，如果有错误的地方恳请指正。

监控范围

引用一下下面这张图来说明：

我们把需要监控的对象分为三层，分别是应用层、系统层和虚拟那一层；这里我们主要关注放在在系统层（CPU、memory、IO等等），以及虚拟层（可能包括容器的OOM，运行时间等等），所以我们主要探讨一下对于这方面的监控。

怎样监控

一般来说，对于Docker的监控，有三种最主要的方式去获取性能指标，分别是：CGroup、Docke命令行以及Docker的API。  
对于CGroup方式，就是通过CGroup的文件来读取这些指标，一般来说在/sys/fs/cgroup目录下面，例如CPU相关的指标/sys/fs/cgroup/cpuacct/docker/$CONTAINER\_ID/cpuacct.stat；这里的CONTAINER\_ID就是容器的ID。  
对于Docker命令行，其实就是通过docker stats来获取：

$ docker stats $CONTAINER\_ID  
CONTAINER       CPU %     MEM USAGE/LIMIT     MEM %     NET I/O             BLOCK I/O  
ecb37227ac84    0.12%     71.53 MiB/490 MiB   14.60%    900.2 MB/275.5 MB   266.8 MB/872.7 MB

和上面的命令行一样，Docker API也能实时采集上面的这些指标，有两种方式来开启开启Docker的API功能，分别是添加这样的参数DOCKER\_OPTS="-H=unix:///var/run/docker.sock -H=0.0.0.0:6732"来分别开启unix sock和http，其中unix sock方式是默认的。

$ echo -ne "GET /containers/$CONTAINER\_ID/stats HTTP/1.1\r\n\r\n" | sudo nc -U /var/run/docker.sock

如果开放了接口，也可以直接通过接口访问，返回是一个很长的json，里面包含了CPU、memory等方面的指标。  
那么对这三种采集方式来说，哪一种是最合适的呢，从排除法来看，命令行的方式获取的指标值比较有限，只能拿到基本的CPU、memory使用状况，更详细的没有，所以这个方案只适合做一个粗略的监控，有对于有些时候的排障来说，可能并不够用；再来看看Docker API的方式，这种方式需要每次发送http请求，而且有多少container就发多少次，这个开销也是不小的，所以这个方案最简单但是我们仍然没有考虑；显而易见，最后我们选择了从CGroup文件的方式来获取，下面我们就来细细说一下需要监控哪些指标，以及怎么来采集。

系统监控

**CPU  
相关的性能指标**

* user CPU：CPU用户进程的时间百分比
* system CPU：CPU执行系统调用的时间百分比
* CPU util：总的CPU使用率
* throttling (count)：容器的CPU被限制的次数
* throttling (time)：容器的CPU使用率被限制的总时间

**采集方式**

$ cat /sys/fs/cgroup/cpuacct/docker/$CONTAINER\_ID/cpuacct.stat  
> user 2441  # docker开启后运行用户进程的时间  
> system 985 # docker开启后运行系统调用的时间

在x86系统中，上面的时间是按10毫秒增加，所以上面的CPU在用户进程上消耗24.41秒，在系统调用上消耗9.85秒。

$ cat /sys/fs/cgroup/cpuacct/docker/$CONTAINER\_ID/cpuacct.usage\_percpu  
> 44154016900  # docker开启后单使用CPU的纳秒（44.15s）

上面是单个CPU使用的时间，如果容器使用的是多核的CPU，那么下面可以获取所有CPU的总的时间：

$ cat /sys/fs/cgroup/cpuacct/docker/$CONTAINER\_ID/cpuacct.usage  
> 44154016900 # 所有CPU使用的总纳秒（44.15s）

对于Throttled，可以在cpu.stat中获取：

$ cat /sys/fs/cgroup/cpu/docker/$CONTAINER\_ID/cpu.stat  
> nr\_periods 565 # 已经过去的执行过的period数  
> nr\_throttled 559 # CPU被限制的次数  
> throttled\_time 11219582971 # CPU被限制的总时间，纳秒为单位 （11.22 seconds）

**指标解读**  
我们知道在Docker中对CPU的限制方式有几种，可以通过--cpu-shares，--cpu-period和--cpu-quota，--cpuset-cpus来配置，具体细节这里不赘述。现在使用最多的方式是--cpu-period和--cpu-quota结合的方式，这时候CPU使用率的上限由两者共同决定，比如说A容器配置的--cpu-period=100000 --cpu-quota=50000，那么A容器就可以最多使用50%个CPU资源，如果配置的--cpu-quota=200000，那就可以使用200%个CPU资源。所有对采集到的CPU used的绝对值没有意义，还需要参考上限。还是这个例子--cpu-period=100000 --cpu-quota=50000，如果容器试图在0.1秒内使用超过0.05秒，则throttled就会触发，所有throttled的count和time是衡量CPU是否达到瓶颈的最直观指标。  
另外，不像传统的host，Docker不需要采集CPU的nice，idle，iowait和irq时间。  
**内存  
相关性能指标**

* Memory：容器的内存使用
* RSS：进程除了缓存之外的内存消耗（包括栈和堆内存，等等）
* Cache memory：内存中的磁盘数据缓存
* Swap：swap使用总量

**采集方式**下面的命令会打印出一大堆的关于内存的信息，可能比你需要的多的多：

$ cat /sys/fs/cgroup/memory/docker/$CONTAINER\_ID/memory.stat  
    cache   
    rss   
    mapped\_file  
    writeback   
    swap   
    pgpgin   
    pgpgout   
    pgfault   
    pgmajfault   
    inactive\_anon   
    active\_anon   
    inactive\_file  
    active\_file   
    unevictable   
    hierarchical\_memory\_limit   
    hierarchical\_memsw\_limit   
    total\_cache   
    total\_rss   
    total\_rss\_huge   
    total\_mapped\_file  
    total\_writeback   
    total\_swap   
    total\_pgpgin   
    total\_pgpgout  
    total\_pgfault   
    total\_pgmajfault  
    total\_inactive\_anon   
    total\_active\_anon   
    total\_inactive\_file   
    total\_active\_file   
    total\_unevictable

虽然上面得到的很多，但是通常我们更关心的核心指标在/sys/fs/cgroup/memory/docker/$CONTAINER\_ID/的其他目录中：

# 总的内存使用: cached + rss   
$ cat /sys/fs/cgroup/memory/docker/$CONTAINER\_ID/memory.usage\_in\_bytes  
# 总的内存使用 + swap的使用  
$ cat /sys/fs/cgroup/memory/docker/$CONTAINER\_ID/memory.memsw.usage\_in\_bytes  
# 内存使用达到限制的次数  
$ cat /sys/fs/cgroup/memory/docker/$CONTAINER\_ID/memory.failcnt  
# 容器被限制使用的内存值   
$ cat /sys/fs/cgroup/memory/docker/$CONTAINER\_ID/memory.limit\_in\_bytes

**指标解读**  
使用的内存可以分解为：  
RSS：RSS本身可以进一步分解为活动和非活动内存（active\_anon和inactive\_anon）。必要时，非活动的RSS内存被交换到磁盘。  
Cache：反映缓存在当前内存中的磁盘上的数据。缓存可以进一步分解为活动和非活动内存（active\_file，inactive\_file）。 当系统需要内存时，可以首先回收非活动内存。  
虽然cache这部分是可以多个容器共享的，但是在Docker中CGroup判断memory.failcnt是否加一，是根据总的内存（RSS+Cache）是否达到memory.limit来决定。所以如果监控到容器的内存使用量一直上升，需要分清是RSS还是Cache导致的增加，如果是RSS的需要看下应用是否有内存泄露，如果是Cache部分，需要看最后是否能释放。  
mem failcnt发生不一定会导致容器OOM，因为有些内存被Cache用到了，OS清理掉一些Cache就没问题了。作为开发者，需要调查下， 给应用划分的Docker内存上限是否合理。因为Cache被清掉就意味着后续有文件读取操作的时候，需要将数据块从磁盘page in到Cache里，如果应用的服务性能比较依赖磁盘上的数据读取性能，就需要关注下。  
另外，在调查性能或稳定性问题时可能有价值的其他指标包括page faults，可以表示分段错误或从磁盘而不是内存中获取数据（分别为pgfault和pgmajfault）。  
**I/O  
相关性能指标**

* I/O serviced：I/O操作的次数
* I/O service bytes：读写的byte数

**采集方式**  
在目录/sys/fs/cgroup/blkio/docker/$CONTAINER\_ID/下有IO相关的指标文件，由于系统的差异，下面大部分文件里面的值都是0，在这种情况下，通常还有两个文件可以工作：blkio.throttle.io\_service\_bytes和blkio.throttle.io\_serviced，它们分别记录了总I/O字节和操作。注意别被文件名误导，这里并不是IO throttle的指标。  
这些文件里前两个数字是主要:次要设备ID，例如blkio.throttle.io\_service\_bytes的输出示例：

253:0 Read 13750272  
253:0 Write 180224  
253:0 Sync 180224  
253:0 Async 13750272  
253:0 Total 13930496

**指标解读**  
块I/O是共享的，所以容器的I/O是没有作限制的，也就没有类似于throttle这样的指标，那么除了上面提到的容器特定的I/O指标之外，跟踪主机的队列和服务时间也是不错的选择。如果容器使用的块设备上的队列长度或服务时间不断增加，容器的I/O将受到影响。  
**网络  
相关性能指标**

* Bytes：网络流量（包括接收和发送）
* Packets：网络包的个数（包括接收和发送）
* Error(receive)：接收错误的数据包个数
* Error(transmit)：传输错误的数据包个数
* Dropped：丢弃的包个数（包括接收和发送）

**采集方式**  
与上面不同的是，网络相关的指标不在CGroup的文件夹下，而是采用平常进程的网络指标采集方式（毕竟Docker也是一个进程），在/proc/下获取：

$ CONTAINER\_PID=`docker inspect -f '{{ .State.Pid }}' $CONTAINER\_ID`  
$ cat /proc/$CONTAINER\_PID/net/dev      
|   Receive                                                |  Transmit  
 face |bytes    packets errs drop fifo frame compressed multicast|bytes    packets errs drop fifo colls carrier compressed  
  eth0:     1296     16    0    0    0     0          0         0      816      10    0    0    0     0       0          0  
    lo:        0      0    0    0    0     0          0         0        0       0    0    0    0     0       0          0

**指标解读**  
同样，网络没有throttle这样的值，衡量时，需要结合网卡的兆数来看。  
**连接数  
相关性能指标**

* established：建立的连接
* close：close状态的连接
* close\_wait：close\_wait状态的连接
* time\_wait：time\_wait状态的连接
* ……

**采集方式**和上面的网络的采集差不多，我们也是通过Docker的pid的方式从/proc/下面去取，具体的文件为/proc/net/tcp和/proc/net/tcp6（如果没有用tcp6可以忽略之）。

$ cat net/tcp  
  sl  local\_address rem\_address   st tx\_queue rx\_queue tr tm->when retrnsmt   uid  timeout inode                                                       
   0: 0100007F:274C 00000000:0000 0A 00000000:00000000 00:00000000 00000000     0        0 10547 1 ffff880426cb0000 100 0 0 10 0                       
   1: 00000000:0016 00000000:0000 0A 00000000:00000000 00:00000000 00000000     0        0 10616 1 ffff880426cb0780 100 0 0 10 0                       
   2: 0100007F:0019 00000000:0000 0A 00000000:00000000 00:00000000 00000000     0        0 11239 1 ffff880426cb1680 100 0 0 10 0                       
   3: 00000000:2742 00000000:0000 0A 00000000:00000000 00:00000000 00000000     0        0 19286 1 ffff880427900780 100 0 0 10 0                       
   4: B42A020A:2742 D014020A:C683 06 00000000:00000000 03:00000CF2 00000000     0        0 0 3 ffff8804216c7d00                                        
   5: B42A020A:2742 D014020A:879D 06 00000000:00000000 03:00001259 00000000     0        0 0 3 ffff8804216c7e00                                        
   6: B42A020A:638C 3602030A:0885 01 00000000:00000000 00:00000000 00000000     0        0 64399466 1 ffff8804256fe180 20 4 30 10 -1                   
   7: B42A020A:2742 D014020A:B04A 06 00000000:00000000 03:000016ED 00000000     0        0 0 3 ffff8804216c6500                                        
   8: B42A020A:2742 D014020A:CC0F 06 00000000:00000000 03:00000D61 00000000     0        0 0 3 ffff8804216c6800                                        
   9: B42A020A:688D 4C02030A:276A 01 00000000:00000000 00:00000000 00000000     0        0 64348606 1 ffff880426cb5280 20 4 30 10 12                   
  10: B42A020A:2742 D014020A:AE80 06 00000000:00000000 03:00001688 00000000     0        0 0 3 ffff8804216c7000                                        
  11: B42A020A:DC39 3502030A:0885 01 00000000:00000000 00:00000000 00000000     0        0 63592428 2 ffff8804256fb480 20 4 30 10 -1                   
  12: B42A020A:851F 4B02030A:276A 08 00000000:00000001 00:00000000 00000000     0        0 12895659 1 ffff8804220b4b00 20 4 28 10 7

其中关注第四列，st就是连接的状态，用下面的数字来表示，具体到和连接状态的映射关系：

   "01": "established",  
    "02": "syn\_sent",  
    "03": "syn\_recv",  
    "04": "fin\_wait1",  
    "05": "fin\_wait2",  
    "06": "time\_wait",  
    "07": "close",  
    "08": "close\_wait",  
    "09": "last\_ack",  
    "0A": "listen",  
    "0B": "closing",  
    "0C": "unknown",

对照映射关系将他们加起来就可以得到对应状态的连接数了。  
**磁盘  
相关性能指标**

* used：磁盘使用量
* used percent：磁盘使用率

**采集方式**  
对于磁盘的采集，我们没有找到一个简便的方法，现在的做法是侵入到容器内部去采集，类似这样的命令：

`docker exec -i $CONTAINER\_PID "df"|grep -v "tmpfs"`

我们来看看直接df之后的结果：

Filesystem           1K-blocks      Used Available Use% Mounted on  
/dev/mapper/docker-253:0-1049953-53ec5aa5f4a669b3a26b19cc2675f5537ba59014419e97324f703cb050152cc6  
                      10475520     44496  10431024   0% /  
tmpfs                 98894428         0  98894428   0% /dev  
tmpfs                 98894428         0  98894428   0% /sys/fs/cgroup  
/dev/mapper/VolGroup01-lv\_root  
                     279403540  38459756 226727800  15% /  
/dev/mapper/VolGroup00-lv\_root  279403540  
                                38459756 226727800  
                                17%  
                                /  
tmpfs                 98894428      2384 98892044  
                      782 99% /  
/dev/mapper/VolGroup00-lv\_root  
                     279403540  38459756 226727800  15% /  
/dev/mapper/VolGroup00-lv\_root  
                     279403540  38459756 226727800  15% /etc/hostname  
/dev/mapper/VolGroup00-lv\_root  
                     279403540  38459756 226727800  15% /etc/hosts  
shm                      65536         0     65536   0% /dev/shm  
tmpfs                 98894428         0  98894428   0% /proc/kcore  
tmpfs                 98894428         0  98894428   0% /proc/timer\_stats  
tmpfs                 98894428         0  98894428   0% /proc/sched\_debug

我们去掉tmpts之后，对剩下的解析，Mounted on在根目录下(/)的device mapper就是我们要的那一行，然后就可以分别得到磁盘的used，available和used percent值了。 **指标解读**  
其实在正确的Docker使用中是不会需要采集磁盘容量的，因为我们对文件的写入应该持久化在宿主机的磁盘上。而且这种采集对资源的消耗很大，如果有需要也要酌情设置采集频率。 **指标解读**  
对于这些不同状态的连接，可以按需去采集。

事件监控

除了上面系统层的监控，我们有时候还需要对容器的事件进行监控，这些事件包括：

attach  
commit  
copy  
create  
destroy  
detach  
die  
exec\_create  
exec\_detach  
exec\_start  
export  
health\_status  
kill  
oom  
pause  
rename  
resize  
restart  
start  
stop  
top  
unpause  
update

我们可以通过docker events来获取到这些事件，该命令支持一个起始时间--since，也支持按不同的条件过滤，包括容器ID，事件类型等等。

$ docker events --filter 'event=stop'  
2018-03-06T00:40:22.880175420+08:00 container stop 0fdb...ff37 (image=alpine:latest, name=test)  
2018-03-06T00:41:17.888104182+08:00 container stop 2a8f...4e78 (image=alpine, name=kickass\_brattain)

每一行就是一个事件，当然我们不会收集这些所有的事件，但一般会包括OOM，stop，destroy这些。

现成的监控方案

当然如果有现成的方案，我们也不需要去重复地造轮子，简单列举几个常用的开源收集吧：

* cAdvisor：Google开发的容器监控指标采集，还支持聚合和一些数据处理；
* Telegraf：Influxdata开发的收集Agent，这是一个通用的采集Agent，当然也支持Docker，另外该公司还提供了一整套监控方案叫做TICK，也欢迎大家去踩坑；
* Prometheus：现在最火的Cloud方面的监控，而且是一整套的解决方案，包括告警、存储等等；

除此之外还有一些收费的方案，例如Datadog、Sensu、Scout等等也提供了另外的选择。

Q&A

Q：既然当前已经存在很多指标监控方案，你们是基于什么考虑要自己写的？A：因为现有的方案都是独立的系统，我们的监控对象可不止容器，而且排查问题的时候可能还需要看宿主机的监控、网络设备的监控等等，我们需要把容器的集成进来；另外用开源的方案不好做定制化。Q：容器发生OOM时，计算的内存是包括Cache+RSS吗？生产环境经常会发生业务容器OOM，可以从那几个方面排查问题，并解决？A：是的，排查问题当然要基于监控，看是否是使用内存一直不释放，我们遇到的OOM一大部分都是应用本身有内存泄漏；这在使用虚拟机的时候没有暴露出来，在用Docker时候资源给得更少了就暴露出来了。Q：是否可以将Pod下所有容器汇总的指标作为Pod的性能指标呢？A：对于Kubernetes来说就更容易一些了，可以通过kubelet API server直接来获取的。Q：当遇到偶发的CPU throttled情况，是否意味着已经开始出现性能瓶颈？A：不是，CPU throttled是在一个period里面CPU的时间片到了限制触发的，如果是job类型的应用，是会偶发cpu throttled，这时候可能不需要关心。Q：现在针对容器的监控方案特别多，也基本上很完善。比如Telegraf采集、普罗米修斯采集等。想问下，你们那边现在的告警是怎么做的？A：告警现在我们更多地配置在应用上，这样反应地最直观。如果要在容器层面做的话，建议对持续的CPU throttling和mem failcnt做告警。Q：请问指标的存储用的是什么数据库？A：之前用的Elasticsearch，现在用的InfluxDB，自己包装实现了一套集群。Q：对于无状态的Java微服务容器，是否有必要进行监控？A：这个最好在应用层去监控，但是在排查问题的时候还是需要容器层面的指标数据。  
Kubernetes 实战培训

本次培训内容包括：Docker容器的原理与基本操作；容器网络与存储解析；Kubernetes的架构与设计理念详解；Kubernetes的资源对象使用说明；Kubernetes 中的开放接口CRI、CNI、CSI解析；Kubernetes监控、网络、日志管理；容器应用的开发流程详解等，点击识别下方二维码加微信好友了解[**具体培训内容**](http://mp.weixin.qq.com/s?__biz=MzA5OTAyNzQ2OA==&mid=2649696800&idx=2&sn=e3fc6b5a34ae2a7532507ce0ed1c6ff1&chksm=88931543bfe49c552c48a367aa2e3211aca8a9829fd3b6d9a71c7c05664d8a88c3ebd01f666d&scene=21#wechat_redirect)。

3月23日开始上课，最后5个名额，点击阅读原文链接即可报名。

[阅读原文](https://mp.weixin.qq.com/s?__biz=MzA5OTAyNzQ2OA==&mid=2649697153&idx=1&sn=546e58faeb8380726ae2e675abf4e754&chksm=889314e2bfe49df42246d351726e54d3d6028ecc5b0570774a97b258cd0cd541e1b4cc1d0b95&mpshare=1&scene=24&srcid=0312cLQ2vMX1wLm61y95tvuX&key=03061ca3b0ce72934191bb9fe84843cf9fed86a165d6f446c5e2e9175821bf89dd97c8380fa1aab7dd4b125336e02af5e01042f3b3ecda6feadb710c0738f983fe5a0f56443ea172a02b98d176de446a&ascene=14&uin=Nzk3NjIyNQ%3D%3D&devicetype=Windows+8&version=6206014b&lang=zh_CN&pass_ticket=lz%2FWjVtDLPWgjZvsnD4SLpRaZaVMVoSg3%2Bv7cF%2Bp3So%3D&winzoom=1##)

阅读 3287

11[投诉](https://mp.weixin.qq.com/s?__biz=MzA5OTAyNzQ2OA==&mid=2649697153&idx=1&sn=546e58faeb8380726ae2e675abf4e754&chksm=889314e2bfe49df42246d351726e54d3d6028ecc5b0570774a97b258cd0cd541e1b4cc1d0b95&mpshare=1&scene=24&srcid=0312cLQ2vMX1wLm61y95tvuX&key=03061ca3b0ce72934191bb9fe84843cf9fed86a165d6f446c5e2e9175821bf89dd97c8380fa1aab7dd4b125336e02af5e01042f3b3ecda6feadb710c0738f983fe5a0f56443ea172a02b98d176de446a&ascene=14&uin=Nzk3NjIyNQ%3D%3D&devicetype=Windows+8&version=6206014b&lang=zh_CN&pass_ticket=lz%2FWjVtDLPWgjZvsnD4SLpRaZaVMVoSg3%2Bv7cF%2Bp3So%3D&winzoom=1##)

[写留言[https://res.wx.qq.com/mmbizwap/zh_CN/htmledition/images/icon/appmsg/icon_edit25ded2.png](javascript:;)](javascript:;)